Technical Report
CMU/SEI-87-TR-038
ESD-TR-87-201

Prototype Real-Time Monitor: Design

Roger Van Scoy
Charles Plinta
Richard D’Ippolito
Kenneth Lee
Michael Rissman

November 1987

Technical Report

CMU/SEI-87-TR-38
ESD-TR-87-201
November 1987

Prototype Real-Time Monitor: Design

Roger Van Scoy
Charles Plinta
Richard D'Ippolito
Kenneth Lee
Michael Rissman

Dissemination of Ada Software Engineering

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler SIGNATURE ON FILE
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Services. For information on ordering, please contact NTIS directly: National Technical Information Services, U.S. Department of Commerce, Springfield, VA 22161.

Prototype Real-Time Monitor: Design

Abstract. This report describes the software design used to implement the prototype real-time monitor (RTM) requirements [D'Ippolito 87]. The design is presented at three levels: system level, object level, and package architecture level. The report concludes with a discussion of the key implementation obstacles that had to be overcome to develop a working prototype: determining system addresses, communicating with an executing application, accessing application memory, converting data into human readable form, and distributed CPU architectures.

Structure of the Document

Chapter 1 of this document gives an overview of the monitoring problem and compares it to other areas where "monitor-like" approaches are used. Chapter 2 provides the high-level architecture for the RTM application system. Chapter 3 examines the objects used to implement the functional needs of the user by proceeding from the lowest to the highest level of abstraction in the system, constantly keeping in mind the needs of the user and building on top of each layer of abstraction. Chapter 4 takes a more "formal" view of the system. Here, object dependency diagrams describe the software architecture (i.e., packaging structure) used to implement the objects described in Chapter 3. Finally, Chapter 5 discusses the key technical issues involved with implementing the RTM, including issues related to performance and use of the RTM in a multiple CPU configuration.

Associated Documents

- *American National Standard Reference Manual for the Ada Programming Language* [Ada 83]
- *Prototype Real-Time Monitor: Requirements* [D'Ippolito 87]
- *Prototype Real-Time Monitor: User's Manual* [Van Scoy 87a]
- *Prototype Real-Time Monitor: Ada Code* [Van Scoy 87b]
- *User's Manual for a Form Generator System in Ada* [Texas Instruments 85a]
- *User's Manual for an ANSI X3.64 Compatible Virtual Terminal in Ada* [Texas Instruments 85b]

Conventions Used in This Document

The conventions used in this document are listed in the left-hand column below; their associated meanings are listed in the right-hand column.

code	Ada language construct
package	Ada package name
subsystem	Ada subsystem
COMMAND	RTM command

Context of Report

The prototype real-time monitor described in this report was built to address two specific technical questions raised by the Ada Simulator Validation Program (ASVP) contractors:

1. How can user tools find, access and display data hidden in the bodies of Ada applications?
2. How can user tools be layered on top of Ada applications?

The prototype is documented by this report because the ASVP contractors had a need for a monitor tool, but did not have the contract resources to develop one. The prototype RTM is intended to be a simple tool which is easily rehostable and extendable. It is not intended to be an example of what a well-documented system should include. Since it was a prototyping effort, no standard documentation or development methods were applied, and no attempt was made to solve all the traditional "monitor" problems.

1. Introduction

A real-time monitor is, in its simplest form, a tool which a software engineer can use to read and write data memory (i.e., variables) in an executing application without halting the CPU. The RTM allows the engineer to do this without requiring any prior knowledge about which memory locations (i.e., variables) need to be operated on. It is real-time because it is intended to be used in conjunction with real-time applications and run in available spare time (in this way it does not perturb the essential timing of the application).

Clearly, this definition contains many elements that are common to other applications:

- The notion of real-time, where the timely execution of the application cannot be disturbed.
- A user interface intended to communicate effectively with a human operator.
- A system interface connecting the RTM to an executing application.
- A transfer of information from the user through the RTM to the application and back to the user.

These concepts are found in other areas under other names. We will highlight two examples of these areas to demonstrate the universal nature of the problem and the general applicability of the solution presented in this report.

1.1. Instructor-Operator Station

The first example is drawn from the flight simulator world. In flight simulators, all training devices possess an instructor-operator station (IOS). This is a user interface device which allows the instructor to control a training exercise. It is used in two ways:

- To set up the configuration for the exercise prior to involving the student.
- To interact with the student while the training exercise is occurring. This allows the instructor to introduce unexpected malfunctions into the simulator and monitor the student's responses.

A pattern is apparent at once: the IOS is a specialized RTM. It has a user interface, allows for the dynamic observance and modification of an application, and is non-interfering (otherwise the exercise is not realistic). The IOS is an example of monitoring when one can predict exactly which parameters will be of interest.[1]

[1] An IOS typically fetches all its parameters on every communication cycle, even though all the data are not needed at any given time.

1.2. Debuggers

A second example is the traditional source-level debugger. Among the characteristics of a debugger are:

- It allows control over the execution flow of the application.
- It allows read and write access to an application’s data objects.
- It is closely tied to a host operating system and target compiler.
- It is very intrusive of the timing of the application.

In light of the functioning of an RTM, the debugger can be viewed as a generalization of the RTM concept. It allows for all the access operations of an RTM, but extends this notion to include specific control over the execution of the application (and as a result loses the ability to not interfere with the real-time nature of an application). The debugger normally has intimate knowledge of how the compiler/linker allocates memory. As we will see later, this information is also needed by the RTM. Both need to interface to the user on a human level and are required to translate data from the internal notation of the computer to the external notation of the user.

Thus, the RTM stands midway between the relative simplicity of an IOS and the complexity of a debugger. The monitor problem is not unique to flight simulators, and the solution presented here is applicable to other domains.

2. System Architecture

The discussion of the RTM starts with the top level view shown in Figure 2-1 (see Appendix B for a complete description of this notation). This figure provides the system-level context in which the RTM operates. In it are four objects:

- User: the human operator controlling some (or all) the other objects in the system.
- Real-time monitor: the system which allows the user to observe the internal function of the application program.
- Application program: the system of interest to the user. The RTM views the application as a cyclic task which has spare processing cycles available for use by the RTM.
- System hardware: the physical devices being driven by the application software (not the computing hardware on which the real-time monitor and application are executing).

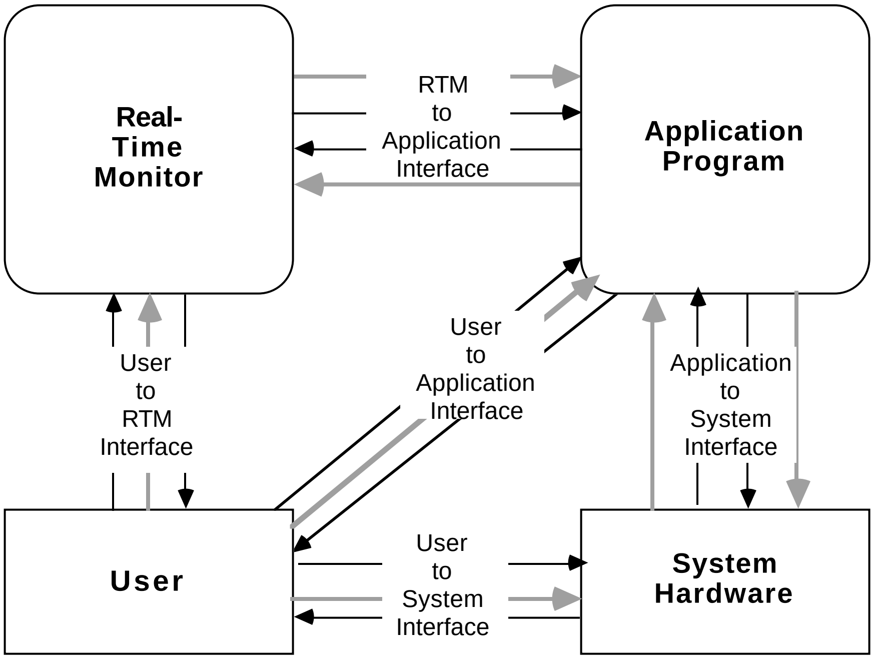

Figure 2-1: System Overview

In addition to these objects are a number of interfaces: user-to-RTM, user-to-application, user-to-system, RTM-to-application, and application-to-system. Of these interfaces, only the user-to-RTM and RTM-to-application are of interest here (the remaining interfaces are not accessible to the RTM). The user-to-RTM interface is the primary focus of the *Prototype Real-Time Monitor: User's Manual* [Van Scoy 87a] and does not enter into the discussions in this report. The interface of most interest in this report, however, is the RTM-to-application interface. The focus is on how the interface is established, how it is manipulated, and how it changes as the system configuration changes.

One additional item to note about Figure 2-1 is that no partitioning of the software among the processors in the system is implied by the diagram. This is a deliberate attempt to isolate the implementation issues of processor configuration from the conceptual (requirements) issues of designing an RTM for Ada applications.

In the real-time monitor portion of Figure 2-1, the RTM is composed of the four classes of objects shown in Figure 2-2:

- Data display objects: handle the RTM side of the user-to-RTM interface.
- Support objects: perform set up and bookkeeping functions for the RTM.
- Data access objects: handle the RTM side of the RTM-to-application interface.
- RTM core object: handles the application side of the RTM-to-application interface.

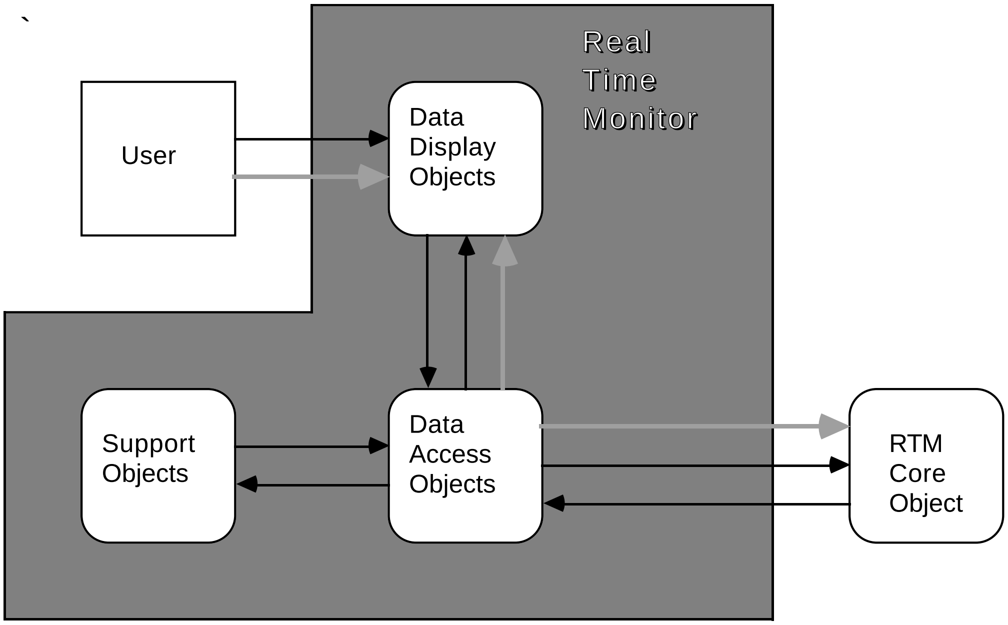

Figure 2-2: Real-Time Monitor Overview

The emphasis of this report is on the data access and RTM core objects since they are central to answering two primary issues:

1. How can user tools find, access, and display data hidden in the bodies of Ada applications?
2. How can user tools be layered on top of Ada applications?

This is not to say that the other objects are unimportant. In this situation, they are simply subordinate to work needed to answer the above questions.

3. Object Architecture

In this chapter, we describe the object architecture of the system by looking at a specific user command and then studying the objects used to implement that command. Specifically, we explain the design by describing:

- all the objects needed to implement the functionality
- the interactions between the objects
- error situations and how they are handled by the objects

3.1. Reading a Variable

The lowest level operation available to the user is reading and displaying a single variable. If the user wished to read the value of the variable **foo** in package **x.y**, the following command would have to be issued:

READ**(name => x.y.foo)**;

The objects necessary to implement this operation (or command) form the framework of the monitor and are shown in Figure 3-1.

3.1.1. Design Objects

Following is an overview of all the objects needed to perform the READ operation:

- *real_time_monitor* is the interface to the user. It takes the user's input, parses the input, and dispatches the command.
- *parameter_manager* manages:
    - verifying the validity of the request
    - extracting the data from the application
    - presenting the data to the user
- *variable_database* builds and manages the collection of all variables accessible to the user. It does this by maintaining a database with all the information needed on any variable accessible to the user through the RTM. This database contains (as a minimum):
    - the Ada variable name (i.e., the full Ada path name)
    - the Ada type
    - the base address of the variable
- *dialogue_manager* hides the details about accessing the data and formatting the raw data into a user-readable form. Using the variable information from the *variable_database*, the *dialogue_manager* is able to extract variable values from the application and (using the *types_manager*) format it so that the user can understand it. Internally, the *dialogue_manager* must maintain, for every active variable (i.e., any variable whose value is requested by the user):
    - the current value of the variable
    - the time tag for the current value

Figure 3-1: Reading a Variable

types_manager knows how to convert:

  - bit strings into character strings
  - character strings into bit strings

- *rtm_core* is an abstraction for the application; it is the only piece of application code which the RTM knows about (or has any control over). Also, being part of the application, it can execute only when the application gives it a slice of time to use. The *rtm_core* operates on the assumption that there is some smallest piece of data which can be accessed (and nothing smaller). It then reads a block of these smallest units and returns the data to the *dialogue_manager*. It performs two primitive operations:

  - get system_storage_unit(s), which returns the value(s) at the specified address(es)
  - put system_storage_unit(s), which writes the specified value(s) into the specified address(es)

- ***standard interface subsystem*** hides the details of parsing user command lines. It uses standard Ada procedure call notation for the command line format. Included in this subsystem is the RTM's command language definition and interpreter (CLI).
- ***virtual terminal subsystem*** hides the device dependencies by exporting a set of terminal independent control operations, that are mapped onto the target display device(s) using UNIX termcap-style definitions (see [Texas Instruments 85b]).
- ***forms management subsystem*** hides the details about how the target display is formatted for output and how the target display is accessed (by treating the screen as a fill-in-the-blank form, see [Texas Instruments 85a]). Included in this subsystem is the RTM's user interface definition.

3.1.2. Object Interactions

With these basic objects in place, we can now look more closely at the interaction among the objects (shown in Figure 3-1) needed to read a variable.

The interaction starts when the user enters the "character data" which form a command line into the ***forms management subsystem*** via the ***virtual terminal subsystem***. These data are sent to the *real_time_monitor* as the "user command line." The *real_time_monitor* then issues a command to the ***standard interface subsystem*** to "parse command line" and waits for the "parser status" signal. If the "parser status" indicates a syntactically (not semantically) legal command line, the command "read" is issued to the *parameter_manager*. If the "parser status" indicated a syntactically incorrect command line a "parser error message" is sent to the user, and the user must start the interaction again.

The next step in processing the READ command is semantic verification. When a command reaches the *parameter_manager*, it is known to be syntactically correct (this check is performed by the ***standard interface subsystem*** when the command is parsed), so the semantic verification process simply consists of:

1. Request the "command arguments," one at a time, from the ***standard interface subsystem***.[2]
2. Query the *variable_database* to determine if the selected "variable is available."

Once that determination is found to be true, the data can be scheduled for extraction. The *parameter_manager* does so by instructing the *dialogue_manager* to "activate variable for data collection." The activation of the variable causes the *dialogue_manager* to request "variable

[2] This is a peculiarity of the command line interpreter software.

information" from the *variable_database* and schedule a read operation in the active read list. When the time for the operation occurs, the *dialogue_manager* takes the variable information from the schedule and sends an "extract data" request to the *rtm_core*. The *rtm_core* will process the request during its next time slice and return the "unformatted variable value" to the *dialogue_manager*. The data are now available to any other process in the RTM.

To summarize what has taken place thus far:

- The user has requested that a variable be read.
- The request has been successfully verified.
- The data have been successfully extracted from the application.
- The data are now sitting in the *dialogue_manager* awaiting further disposition.

The fact that the data of interest have been moved from one area of memory (or one processor) to another is of little interest to the user. What is now needed is for the information to be presented to the user. This presentation occurs when the *parameter_manager* requests the "formatted variable value" from the *dialogue_manager*. The *dialogue_manager* has the data in an internal format, but does not know what to do with them. To fulfill the request, the *dialogue_manager* passes the "unformatted variable value" to the *types_manager*, which converts the bit string into a "formatted variable value" for the *dialogue_manager* to send back to the *parameter_manager*.

Once the data are converted into a human-readable form, the *parameter_manager* sends the "display data for variable" to the ***forms management subsystem*** for presentation to the user. The command has now been successfully executed, and the RTM is ready to process another command.

3.1.3. Error Processing

There are two classes of errors which can arise in the course of this processing. First, the variable may not be accessible to the RTM (i.e., it does not exist in the database of available variables). In this situation, the *parameter_manager* informs the user that the variable is not accessible and the processing is complete. Second, the variable may be accessible, but an error may occur in the *rtm_core* when the data are accessed. Here, the *parameter_manager* informs the user that the variable could not be read from application memory and the processing is complete.

3.2. Reading a Page of Variables

The user can now access any available variable in the system by issuing READ commands to the RTM. Powerful as this is, it is a tremendous burden to use the READ command to repetitively examine one variable, let alone a group of variables. Clearly, there is now a need for a higher-level abstraction. This abstraction is called a "page." A page is simply a collection of individual variables which are extracted and displayed as a group. Using a page, one command can produce a wealth of information. Still, if this is a repetitive request, the user cannot control the regularity of the extraction process. Thus, we introduce an update_rate, which informs the RTM that a page is to be processed and displayed at a periodic rate determined by the user. With a page and an update_rate, the user has a powerful abstraction for monitoring an application.

3.2.1. Design Objects

The objects used to implement reading a page of variables are the same as those used in Section 3.1 to read a single variable, with one exception: the *parameter_manager* is replaced by the *page_processor* (shown in Figure 3-2, which is structurally equivalent to Figure 3-1), the differences are highlighted by bold typeface.

The *page_processor* object manages:

- verifying the validity of the request
- extracting the data from the application
- presenting the data to the user

3.2.2. Object Interactions

There are three distinct sequences of interactions which occur in Figure 3-2, each initiated by a different user command. To start with, the interactions needed to create a page begin with the user issuing the command:

```
EDIT ();
```

The object which implements the editing or building of a page is the ***forms management subsystem***. The ***forms management subsystem*** is essentially an editor which allows the user to construct a display template by full-screen editing and later allows the RTM to place the collected data in this template. The editing process is fully described in the *Prototype Real-Time Monitor: User's Manual* [Van Scoy 87a]. Since the ***forms management subsystem*** has no knowledge about variables in the *variable_database*, there is a companion command to EDIT which can be used. The command:

```
CHECK (page => example);
```

can be used after the EDIT command to perform error checking (described below) on a page without actually starting active data collection on that page.

Once a page has been created, that page must be invoked for processing. In this case, the command looks like:

```
START (page => example, update_rate => 0.5);
```

Again, when a command reaches the *page_processor*, it is known to be syntactically correct, so the semantic verification processing consists of:

1. Request the "command arguments," one at a time, from the ***standard interface subsystem***.
2. Request the "page definition data" from the ***forms management subsystem***.
3. Add the new page to the list of active pages.

The *page_processor* takes the "page definition data" and queries the *variable_database* to determine if the "variable is available" for each variable defined on the page. Once that determination is found to be true, each variable is scheduled for extraction. Again, this is done by instructing the *dialogue_manager* to "activate variable for data collection" for each variable. The activation

Figure 3-2: Displaying a Page of Variables

of the variable causes the *dialogue_manager* to request "variable information" from the *variable_database* and schedule a read operation in the active read list. When the time occurs for the page to be updated, the *dialogue_manager* takes the variable information from the schedule and sends an "extract data" request for each variable to the *rtm_core*. The *rtm_core* will process the requests during its next time slice and return the "unformatted variable value" to the *dialogue_manager*. The data are now waiting in the *dialogue_manager* for the next scheduled display update, at which time the *page_processor* requests the "formatted variable value" for each variable on the page. The data and control information are combined to form the "display data for page" which is sent to the ***forms management subsystem*** for presentation to the user. Since the START command has an associated update_rate, this processing will continue until terminated by the user.

The final processing sequence occurs when the user terminates an active display. To accomplish this, a command like:

```
STOP (page => example);
```

is issued. Here, the processing consists of:

1. Request the "command arguments" from the ***standard interface subsystem***.
2. Request the *dialogue_manager* to "deactivate variable" (where it removes the variable from the active read list and releases its data storage) for each variable on the page.
3. Remove the page from the list of active pages.
4. Remove the page from the display device.

This covers the complete cycle of page operations: building a page to displaying a page to terminating a page.

3.2.3. Error Processing

There are four classes of errors which can arise in the course of this processing. First, the requested page may not be accessible. In this situation, the *page_processor* informs the user and the processing is complete. Second, a variable may not be accessible to the RTM (i.e., it does not exist in the database of available variables). In this situation, the *page_processor* deletes the variable from the page and informs the user that the variable is not accessible. Processing then continues on any remaining variables defined for the page. Third, a variable may be accessible, but an error may occur in the *rtm_core* when the data are accessed. Here, the *parameter_manager* informs the user that the variable could not be read from application memory and processing continues on any remaining variables. Fourth, the limit number of active pages may have been reached. In this case, the processing never starts and the user is informed of the error. It is then up to the user to STOP a currently active page and START the desired page.

3.3. Writing a Variable

The final operation available via the RTM is the ability to modify application memory. This requires revisiting the first object, the *parameter_manager* (shown in Figure 3-3, which is structurally the same as Figures 3-1 and 3-2, with differences highlighted by boldface type), and considering some additional functionality. Suppose the user wished to change the value of the variable **foo** in module **x.y** and the following command is issued:

Set (name => x.y.foo, value => 10);

3.3.1. Design Objects

The objects in the system are the same ones discussed in Section 3.1. The difference occurs in the low-level processing needed to implement the operation.

3.3.2. Object Interactions

After the "user command line" has been successfully parsed, the next step is semantic verification. When a command reaches the *parameter_manager*, it is known to be syntactically correct, so the semantic verification process simply consists of:

1. Request the "command arguments," one at a time, from the ***standard interface subsystem***.
2. Query the *variable_database* to determine if the selected "variable is available."

If that determination is found to be true, the data can be scheduled for modification. The *parameter_manager* does so by instructing the *dialogue_manager* to "activate variable for data deposit." The activation of the variable causes two actions to occur:

1. The *dialogue_manager* passes the "formatted variable value" to the *types_manager* and receives "unformatted variable value" in exchange (which is logged internally).
2. The *dialogue_manager* requests "variable information" from the *variable_database* and schedules a write operation in the active write list.

When the time for the operation occurs, the *dialogue_manager* takes the variable information from the schedule and sends a "deposit data" request to the *rtm_core*. The *rtm_core* will process the command request during its next time slice and return the "deposit status" to the *dialogue_manager*. The "deposit status" is then returned to the *parameter_manager* for subsequent presentation to the user.

3.3.3. Error Processing

There are three classes of errors which can arise in the course of this processing. First, the variable may not be accessible to the RTM (i.e., it does not exist in the database of available variables). In this situation, the *parameter_manager* informs the user that the variable is not accessible and the processing is complete. Second, the variable may be accessible, but an error may occur in the *rtm_core* when the data are accessed. Here, the *parameter_manager* informs the user that the variable could not be written to application memory and the processing is complete. Third, the user may have entered data in an inappropriate format for the variable (i.e., attempting to assign the value 10z instead of the integer 100). In this case, the input is rejected and the user must reenter the command.

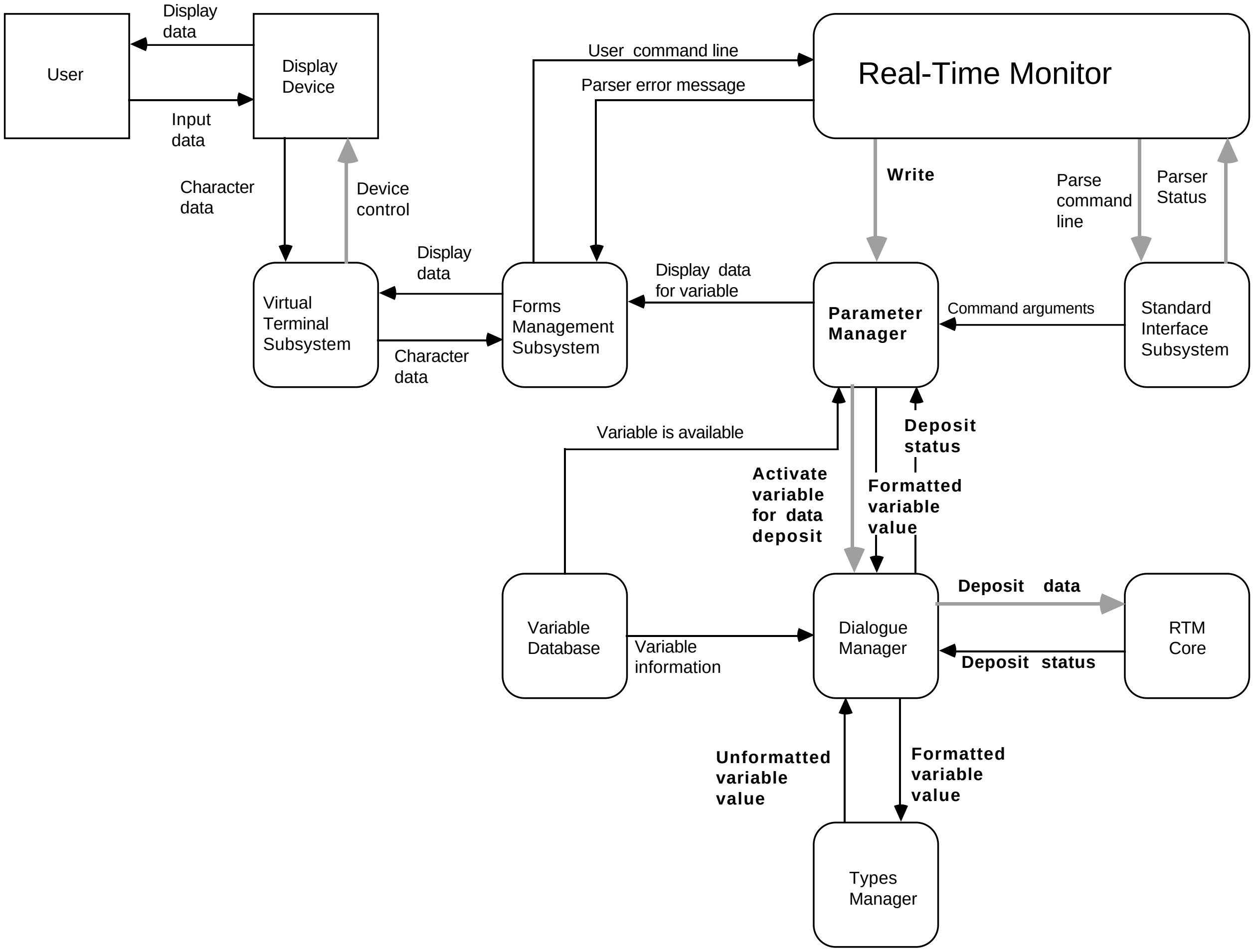

Figure 3-3: Writing a Variable

4. Package Architecture

This chapter provides a guided tour through the architecture of the RTM, giving an overview of the prototype and briefly explaining the purpose and key abstractions each package encapsulates. The top level of the RTM is shown in Figure 4-2 (see Appendix A for a complete description of this notation), with the successive levels shown in Figures 4-3 through 4-7.

4.1. Prototype RTM

The complete picture of the capabilities and usage of the RTM are found in the *Prototype Real-Time Monitor: User's Manual* [Van Scoy 87a]. Here, we briefly give an overview of the prototype. The commands implemented in the prototype are:

- EDIT ();
- CHECK (name => <page>);
- QUIT ();
- READ (name => <variable>);
- SET (name => <variable>, value => <number|string>);
- START (name => <page>, update_rate => <time>;
- STOP (name => <page>);

The following restrictions have been imposed on the prototype:

- Single display device with VT100 terminal characteristics:
    - 80 columns by 24 lines
    - keyboard input only
- No simultaneous input and output to the display device (i.e., screen updating halts during user command entry).
- Integer, float, and enumeration data types[3] only.
- Generation of the variable database and type conversion routines is the user's responsibility.

This prototype was built using as much existing software as possible. Figure 4-1 gives the statement count and total line count for the RTM development task. Only the 13% list for the ***RTM subsystem*** (shown in line 1 of Figure 4-1) is newly developed software. The remainder of the code is from the Ada Software Repository and reused without modification.

[3] **access** type variables can be used to monitor the underlying object.

Subsystem	Statements[4]	Total Lines[5]
RTM	1218	8058
Forms	2293	6894
Virtual Terminal	2421	6269
Parser	1896	9600
Utilities	1522	6154
Totals	9350	36999

Figure 4-1: Sizing Figures for the Prototype

4.2. Real-Time Monitor

The *real_time_monitor* package (shown in Figure 4-2) holds the entire structure together. It functions primarily as a cyclic executive for the RTM:

- Takes user input using the ***forms management subsystem*** and the ***virtual terminal subsystem***.
- Parses the input using the ***standard interface subsystem***.
- Dispatches commands for execution using the *page_processor* and *parameter_manager* packages.
- Periodically updates the display device using the ***forms management subsystem*** and the ***virtual terminal subsystem***.

4.3. Page Processor

The *page_processor* package (shown in Figure 4-3), as discussed earlier, encapsulates the page abstraction. It is solely responsible for managing the interface to the page objects created by the user. It does this by hiding all the details about how a page is:

- Invoked (**Start_Page**).
- Periodically updated (**Update_Page** and *dialogue_manager*) and displayed (using the ***forms management subsystem***).
- Terminated (**Stop_Page**).
- Represented internally (**Setup_Page** and **Check_Page**).
- Checked for consistency using the *variable_database* package.

[4] The statement count is produced using another Ada Software Repository utility, Pager, which counts Ada statements (excluding comment lines) rather than semicolons.

[5] All lines in all files.

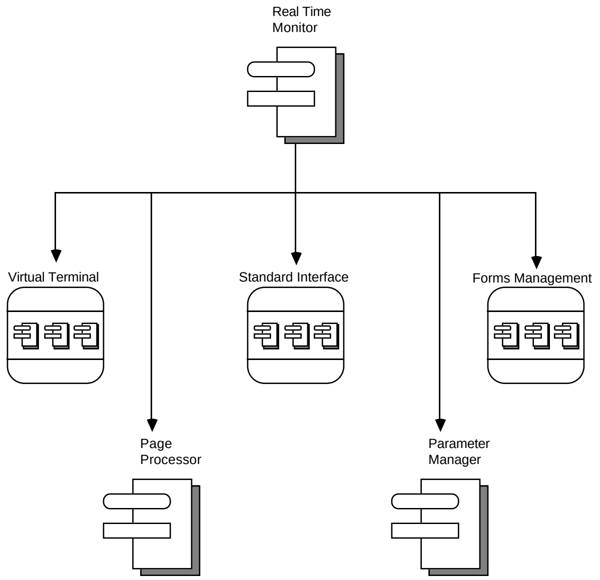

Figure 4-2: RTM Object-Dependency Diagram

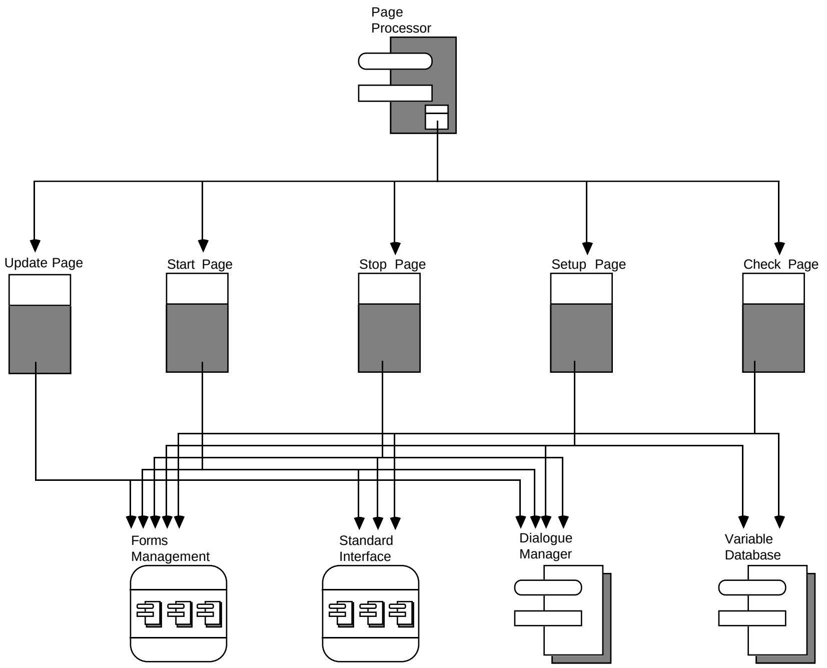

Figure 4-3: Page Processor Object-Dependency Diagram

4.4. Parameter Manager

The *parameter_manager* package (shown in Figure 4-4) performs the same basic functions as the *page_processor* package. The major difference is that it manages the interface to single variable operations. It does this by encapsulating:

- How a variable is read (**Read**).
- How a variable is written (**Set**).
- How a request is verified and processed using the *dialogue_manager* package.
- How a request is displayed using the ***forms management subsystem***.

4.5. Variable Database

The *variable_database* package (shown in Figure 4-5) is the heart of the RTM. Without this abstraction, nothing else in the system can function. It is responsible for knowing which variables are accessible to the user (via the information it obtains from the *library_interface*). The *variable_database* is not responsible for generating the database information (see the *library_interface* below). Its functions include:

- Building the structure which holds the information (**Initialize_Database**).
- Managing the structure.
- Providing the interface needed to access the structure (**Find**).

This allows the rest of the system to specify the minimum amount of information needed for the RTM to function and isolate itself from how the information is generated and controlled.

4.5.1. Library Interface

The *library_interface* package (shown in Figure 4-5) is actually responsible for generating the information which goes into the variable database. There are several reasons for this split between the *variable_database* and the *library_interface*. First, the *variable_database* need not have any knowledge of the items in the structure which it manages. Second, it allows for further isolation of the system-dependent parts of the RTM. Clearly, most systems cannot supply the information required to construct the database. Thus, the ability to build this database is system dependent. The more information the *library_interface* can provide to the *variable_database*, the more flexibility the user has in monitoring capability.

4.6. Dialogue Manager

If *variable_database* is the heart of the system, then the *dialogue_manager* package (shown in Figure 4-6) is the soul of the system. It manages the interface between the RTM and application. It hides all the details related to reading and writing application memory, the scheduling of these operations, and the conversion of bit strings extracted from the application into character strings for the user.

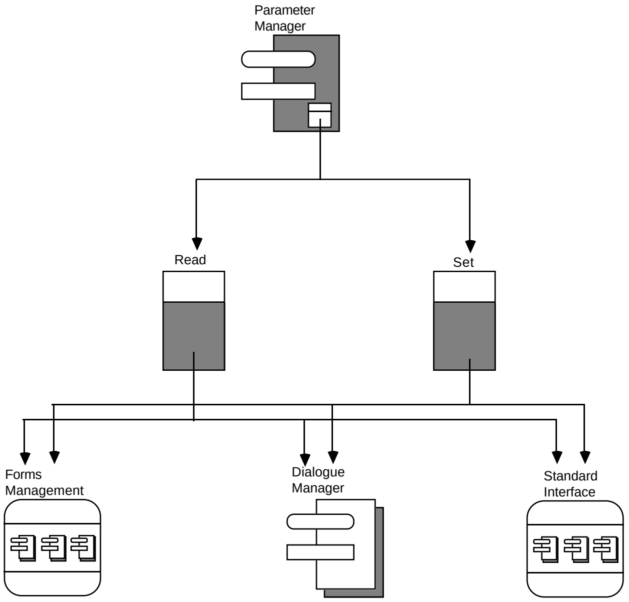

Figure 4-4: Parameter Manager Object Dependency Diagram

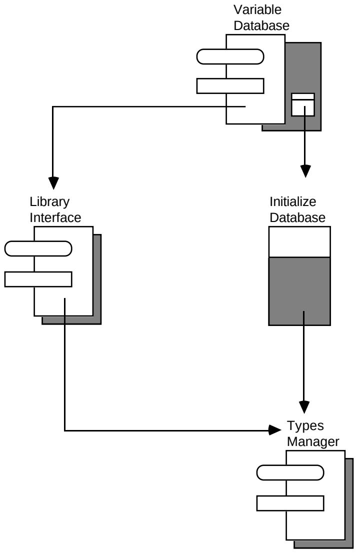

Figure 4-5: Variable Database Object-Dependency Diagram

4.6.1. Collect Data

While the *dialogue_manager* does the scheduling of the read and write operations, it is the responsibility of the *collect_data* package (shown in Figure 4-6) to:

- Format the commands (**Build_Rtm_Core_Commands**).
- Communicate with the *rtm_core* package (**From_Application**).
- Store the results (**Retrieve_Rtm_Core_Results**).

4.6.2. RTM Core

The *rtm_core* package (shown in Figure 4-6) is the actual agent which reads and writes application memory. As noted previously, this is the abstract application with which the RTM communicates. In a real system, this package becomes part of the application and provides the interface needed by the RTM. Thus, it hides all the details related to actually manipulating application memory. A more detailed discussion of the internal functioning of this package can be found in Chapter 5.

4.6.3. Sysgen

The *sysgen* package (shown in Figure 4-6) provides the ability to partition the software based on the available hardware suite (discussed in Section 5.5) and to control the timing of the resulting system. Using the parameters in this package, the user can tailor the RTM to match the available resources of the system. This tailoring is fully discussed in the *Prototype Real-Time Monitor: User's Manual* [Van Scoy 87a].

4.6.4. Address Generator

The *address_generator* package (shown in Figure 4-6) is responsible for supplying the address abstraction used by the RTM. It supports this function by:

1. Exporting the abstract address type, **Address_Representation**.
2. Exporting the **Compute_Address** function to generate abstract addresses.

This package is responsible for hiding the manner in which system addresses are generated, thus allowing for different address-generation schemes to be used interchangeably.

4.7. Types Manager

Finally, the lowest-level package in the RTM is the *types_manager* package (shown in Figure 4-7). Due to the nature of the interface between the *dialogue_manager* and the *rtm_core*, the data from the the application come across the interface as a bit string, with no attempt at interpretation. The result is that a data conversion must occur before the results can be presented to the user. The *types_manager* is the object that knows how to map bit strings into character strings. This object allows the RTM to be insulated from these low-level details and thus improves the portability of the system (since the underlying bit patterns of a value will probably change from machine to machine).

4.7.1. Conversions

To ease the burden of converting all the variants on the base Ada types, the *conversions* package includes three generic conversion packages (based on **Text_IO** utilities). These generics convert arbitrary bit strings into character strings. These routines also do the low-level bit shifting needed when using the RTM in a multiple, heterogeneous CPU configuration.

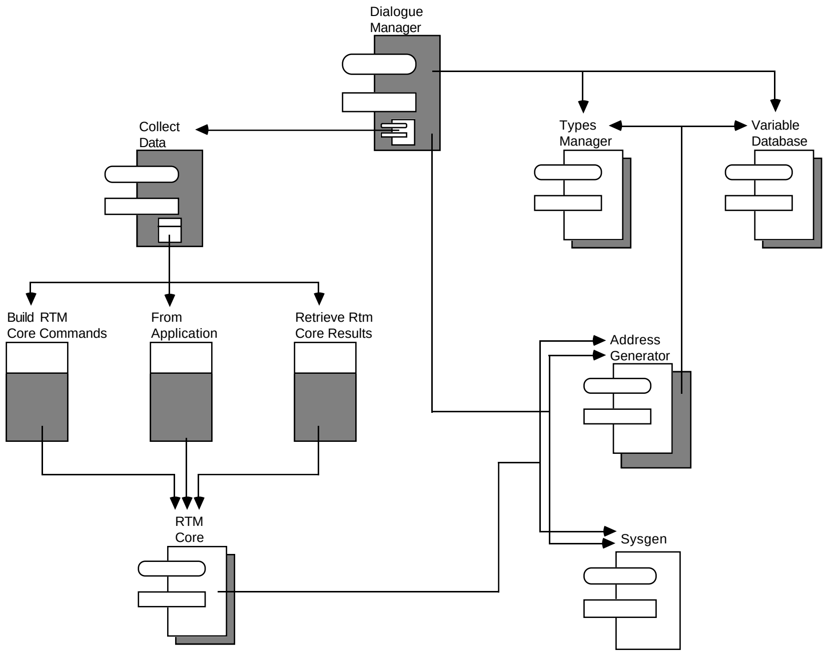

Figure 4-6: Dialogue Manager Object-Dependency Diagram

Figure 4-7: Types Manager Object-Dependency Diagram

5. Implementation

While this chapter does not consider the entire RTM implementation, it does discuss the key implementation obstacles overcome in implementing the RTM. These are:

- Generating the variable database and determining system addresses.
- Communicating interface to the application.
- Accessing application memory.
- Converting data into human readable form.
- System architecture.

Each of these areas is important and will be discussed in detail below.

5.1. Variable Database and System Addresses

As noted in Chapter 4, the *variable_database* is the foundation of the monitor. Without the ability to determine if a variable is in the application and then determine its address, the RTM cannot function. This applies equally to the symbolic debugger (which gets this information from the compiler and linker) and IOS (which forces the data of interest to known addresses), discussed in Chapter 1. Thus, the RTM needs the following:

- Compiler/linker output: variable, address, type.
- Type information: length (in bits), record formats, component offsets, indirection (access type) indications.
- Computation routines for address of record/array element accesses.
- Computation routines for dynamic objects (local variables, loop variables, etc.).

The RTM isolates these system dependencies by using a database of available variables (structured as an ordered binary tree), a database of available types, and an address computation function that processes the type and variable information in the databases to produce a system address. A complete discussion of the approach used in the prototype to generate its variable database can be found in the *Prototype Real-Time Monitor: User's Manual* [Van Scoy 87a].

5.2. Communications Interface

Given that addresses can be generated for application data objects, the next consideration is how the user's commands are communicated to the application. As discussed previously, the *rtm_core* package is the object that ultimately affects the application. The interface between the RTM proper and the *rtm_core* (which is synonymous for the application) is composed of two buffers: a command buffer and a data buffer, shown in Figure 5-1. The command buffer is composed of a sequence of commands. Each command contains the fields:

1. Status/operation to be executed (some of these are *rtm_core* control operations):

    a. buffer available
    b. results available
    c. deposit

d. extract
e. end of buffer

2. Address of the data to operate on:

   a. base address
   b. address offset
   c. indirection indicator

3. Amount of data to be read/written.
4. Location in the data buffer where deposit data reside or extract data are to be stored.

```
 package Rtm_Core is
   subtype  Buffer_Range is  Integer range  1..Sysgen.Core_Buffer_Size;
   type  Buffer_Entry_Representation is  record
     Command: Rtm_Core_Command_Representation := End_Of_Buffer;
     Data_Address: Address_Generator.Address_Representation;
     Data_Count: Buffer_Range;
     Data_Location: Buffer_Range;
   end  record ;

   Command_Buffer: array  (1..Buffer_Range'Last) of
     Buffer_Entry_Representation := (others  =>
        (End_Of_Buffer,Address_Generator.Null_Address,1,1));

   Data_Buffer: array  (1..Buffer_Range'Last) of
     Sysgen.Smallest_Unit := (others  => 0);
end  Rtm_Core;
```

Figure 5-1: Communications Interface Definition

This command structure allows the *rtm_core* processing to be extremely simple. Only the deposit and extract operations (which are discussed in detail later) require any significant processor time. Also, all the components of the **command_buffer** are integers or subtypes of integers. This allows the interface to the *rtm_core* to be easily separated from the rest of the RTM and placed with the application on a separate processor (discussed in detail later).

5.3. Accessing Application Memory

As noted previously, the deposit and extract operations are the only ones which require processor time. To isolate the RTM from the application, Ada language dependencies, and system architecture constraints, all addresses are treated internally as records containing:

- base address (as integer)
- address offset (as integer)
- indirection indicator

In this way, application memory is viewed as a block of elements of type **smallest_unit** (or bit string), which is defined as an adjustable parameter in *sysgen*. By treating the base address and offset as integers, the RTM need not deal with differences in address space between the RTM and application. Also, by viewing all application data as a single abstract type, we can treat the data as strings of bits without any knowledge of the underlying data type.

To illustrate how this information can be used, see Figure 5-2. This figure shows a segment of application memory where three variables — foo, blark and ratio — reside. For the RTM to read the value in the variable foo, for instance, it must somehow determine that the data reside at address 164 in application memory. To do this, the RTM obtains the address of foo from **address_generator.compute_address**. Using the type information obtained from *types_manager*, commands are formatted to instruct the *rtm_core* to extract the value of foo.

Figure 5-2: Application Memory

The key to the *rtm_core* is the manner in which these addresses are manipulated. The first code fragment, shown in Figure 5-3, performs the setup needed before processing an address. In this fragment:

1. A type is created which accesses an object of type **smallest_unit**.
2. A data object is created which accesses a value of type **smallest_unit**.
3. **Unchecked_conversion** is instantiated to convert an integer into an access value for an object of type **smallest_unit**.

This lays the groundwork for actually using the integer to access data objects in the application.

```
with Unchecked_Conversion;
type Value_Pointer is access Smallest_Unit;
The_Value: Value_Pointer;
function Get_Address is new Unchecked_Conversion
      (Source => Integer,
       Target => Value_Pointer);
```

Figure 5-3: Setup Code Fragment

The code shown in Figure 5-4 is used to extract a value from application memory. Here, the RTM:

1. Computes the actual address of the data object using integer arithmetic.
2. Converts the integer into a pointer to a value of type **smallest_unit**.

3. Uses the access variable to move the data from the application memory into the **data_buffer**, without any data conversion taking place.

This is the key: the bit pattern in application memory must be moved into the communications area without any alterations by Ada. Otherwise, the data value extracted from the **data_buffer** later in the processing will not be the original bit pattern. This is achieved by creating a pointer to the **smallest_unit** type (even though the actual bit pattern at the address corresponds to a different type) and manipulating the data as if it were actually of type **smallest_unit**.

```
  procedure  Extract_Data (Data_Address: in  Integer;
                   Command_Number: in  Buffer_Range) is
--|*****************************************************************
--| Description:
--|   Moves the data from application memory into data_buffer passed
--|   back to the RTM.
--|
--| Parameter Description:
--|   data_address    ->  The computed address of the desired data.
--|                   In the case of a mulitple unit read, this
--|                   is the address of the first unit in the block.
--|   command_number  ->  Command being processed in the command_buffer.
--|
--| Notes:
--|     none
--|*****************************************************************
  Next_Address: Integer := Data_Address;
  The_Value: Value_Pointer := Get_Address(Next_Address);
  Data_Offset: Buffer_Range renames  Command_Buffer(Command_Number).Data_Location;
  begin
    for  Next_Data_Position in  0..Command_Buffer(Command_Number).Data_Count-1 loop
      Data_Buffer(Next_Data_Position + Data_Offset) := The_Value.all ;
      Next_Address := Next_Address + 1;
      The_Value := Get_Address(Next_Address);
    end  loop ;
  end  Extract_Data;
```

Figure 5-4: Extract_Data Procedure

The final procedure involves writing data into application memory. This is shown in Figure 5-5. The RTM:

1. Computes the actual address of the data object using integer arithmetic.
2. Converts the integer into an access to a value of type **smallest_unit**.
3. Moves the bit pattern in **data_buffer** into application memory using the access variable, again without any data conversion taking place.

This is simply the inverse of the extraction operation discussed above. Taken together, this code allows the RTM to read and write application memory (without any detailed knowledge about the underlying types being manipulated).

```
   procedure  Deposit_Data (Data_Address: in  Integer;
                    Command_Number: in  Buffer_Range) is
--|****************************************************************
--| Description:
--|   Moves the data from the data_buffer passed by the RTM into
--|   application memory.
--|
--| Parameter Description:
--|   data_address    ->  The computed address of the desired data.
--|                   In the case of a mulitple unit read, this
--|                   is the address of the first unit in the block.
--|   command_number  ->  Command being processed in the command_buffer.
--|
--| Notes:
--|     none
--|****************************************************************
   Next_Address: Integer := Data_Address;
   The_Value: Value_Pointer := Get_Address(Next_Address);
   Data_Offset: Buffer_Range renames  Command_Buffer(Command_Number).Data_Location;
   begin
     for  Next_Data_Position in  0..Command_Buffer(Command_Number).Data_Count-1 loop
       The_Value.all  := Data_Buffer(Next_Data_Position + Data_Offset);
       Next_Address := Next_Address + 1;
       The_Value := Get_Address(Next_Address);
     end  loop ;
   end  Deposit_Data;
```

Figure 5-5: Deposit_data Procedure

5.4. Type Conversions

5.4.1. Top-Level Organization

The final link in the chain for data coming from the application is conversion to a human understandable form. There are several objectives:

- Bit strings (or blocks of **smallest_unit**s) coming from the application have to be converted into human readable character strings.
- All the details about performing the low-level bit manipulations and conversions have to be hidden. This is done by using the two procedures shown in Figure 5-6:
    - **Convert_Value_To_String**, which takes the bits and makes the character string for the user.
    - **Convert_String_To_Value**, which takes a user-entered value and makes the application a bit string.
- All the details about the internal structure of types and what types exist within the system need to be hidden. This is accomplished by the two procedures in Figure 5-7, namely:
    - **Find**, which takes the name of a type and returns an internal identifier for that type.
    - **Get_Type_Information**, which takes a type identifier and returns that information about a type that must be available to the outside world.

This top-level organization provides sufficient abstraction and hiding for our purposes. Now, we look at the low-level implementation which actually converts the bit stings into character strings.

```
   procedure  Convert_Value_To_String (Data_Type: in  Valid_Rtm_Type;
                              Raw_Data: in  System.Address;
                               Number_Of_Characters: in  Integer;
                               The_Value: out  String);
--|*****************************************************************
--| Description:
--|   This module converts from the internal representation used
--|   by the RTM in storing variable values into strings that
--|   are displayable to the user.
--|
--| Parameter Description:
--|   data_type   ->The Ada data type of raw data.
--|   raw_data    ->  The address of the binary bit string to convert.
--|     number_of_characters -> The number of characters needed in the
--|                         value string.
--|     the_value   ->  A string containing the displayable value.
--|*****************************************************************

   procedure  Convert_String_To_Value (Data_Type: in  Valid_Rtm_Type;
                              Raw_Data: in  System.Address;
                               The_Value: in  String);
--|*****************************************************************
--| Description:
--|   This module converts from the string entered by the user
--|   into the internal representation used by the RTM and in
--|   storing values.
--|
--| Parameter Description:
--|   data_type   ->  The Ada data type of raw data.
--|   raw_data    ->  The address of the binary bit string to convert.
--|   the_value   ->  The string whose value the user wishes deposited into
--|                  application memory.
--|*****************************************************************
```

Figure 5-6: Data Conversion Interface

5.4.2. Low-Level Implementation

The code examples shown here all deal with converting bit strings into integer character strings. The same concepts and techniques are used to convert floats and enumerations[6] to character strings. An inverse approach is used to convert from character strings into bit strings. The basic approach to converting the bit strings is similar to that used in accessing the application data, relying heavily on **access**[7] types and **Unchecked_Conversion**.

All the actual low-level conversion (for integers) is done by the generic package *convert_integers*, shown in Figure 5-8. This particular generic takes in the type of the source and a routine which converts the target processor's data representation into the host processor's data representation. To illustrate these points, an instantiation of *convert_integers* is shown in Figure 5-9.

The **Make_String** procedure uses the **Target_Conversion** routine to map the target data into the host's form; the value is then converted to a string using the services available in **Text_IO**.

[6] For enumeration conversions, the body of *types_manager* package must have a definition of each enumerated type.

[7] Care must be taken in the use of the **'address** attribute since it may need to be adjusted to obtain the true address of the data.

```
--
--  Type identifier, used externally to refer to a named type.
--
   type  Valid_Rtm_Type is  private ;

   function  Find (Name: in  String) return  Valid_Rtm_Type;
--|****************************************************************
--| Description:
--|   This module is the lookup entry used to locate legal types,
--|   It maps data obtained from the library_interface into types
--|   which the types_manager can convert.
--|
--| Parameter Description:
--|   name    ->  The name of the Ada type associated with
--|             a variable.
--|   return  ->  The internal Identifier used to refer
--|             to the type.
--|****************************************************************

   procedure  Get_Type_Information (Type_Identifier: in  Valid_Rtm_Type;
                                    Type_Length: out  Integer;
                                    Indirection_Indicator: out  Boolean);
--|****************************************************************
--| Description:
--|   This module takes a type identifier and returns detailed
--|   information about the structure of the type to the caller.
--|
--| Parameter Description:
--|   type_indentifier    ->  Identifier of the type about which
--|                        information is needed.
--|   type_length ->  The size of the underlying type in the
--|                 size of the storage units used by the RTM
--|                 (i.e. smallest_units).
--|   indirection_indicator   -> A boolean flag which when
--|                             true => an access type
--|                             false => any other type
--|****************************************************************

private

   type  Valid_Rtm_Type is  new  Integer;
```

Figure 5-7: Type Information Interface

The **Default_Integer_Conversion** procedure is a dummy routine setup for the single CPU configuration of the RTM. In this case, it simply takes the address of a bit string and returns an integer value. In a multiple CPU configuration, this procedure might be called upon to convert from the application processor's integer representation to the host processor's integer representation. The generic can now be instantiated with this conversion routine and perform its processing without any knowledge of the differences in numeric representation between the various processors in the system. Using this service, **Convert_Value_To_String** can now accept any bit string from the application and convert it to a character string for the user. By adding additional functionality, these services could also produce octal, binary, or hexadecimal output.

```
with  System;
--  Need the type "Address".
--
 package Convert_Integers is

generic
--
--  Default Width of  The Generated Character Strings.
   Width: Positive := 15;
--
--  Integer type  Source, This is  The Host Machine'S type
   type  Source_Representation is  range  <>;
--
--  Low Level Conversion Routine Needed To Convert From The Target
--  Representation To The Host Representation of  The Source type
--  (Referred To As Source_Representation)
   with  function  Target_Conversion (Raw_Value: in  System.Address)
      return  Source_Representation;
End Convert_Integers;

package   generic package body Convert_Integers is
procedure  Make_String  (Raw_Value: in  System.Address;
                  Field_Size: in  Integer;
                  Value: out  String) is
--|*****************************************************************
--| Description:
--|    Make_string takes a binary bit string and converts it into
--|    an integer character string.  It does this by using
--|   target_conversion to map the target bit representaion of and
--|   integer into the host version of an integer and then
--|   uses text_io to convert the bits into an integer character string.
--|
--| Parameter Description:
--|    raw_value   ->  The address of the binary bit string to be
--|                 converted.
--|    field_size  ->  The number of characters needed in the output
--|                 string.
--|    value   ->  The character image of the binary bit string, as
--|              an integer.
--|
--| Notes:
--|    none
--|*****************************************************************
begin
   if  Width > Field_Size then
      Value(1..Field_Size) := (1..Field_Size => '*');
   else
      Internal_Io.Put (To => Value(1..Width),
                   Item => Target_Conversion(Raw_Value));
   end  if ;
exception
   when  others  => RAISE ;
end  Make_String;
end  Convert_Integers;
```

Figure 5-8: Convert_integers Package

```
    type  Integer_Pointer is  access  Integer;
    function  Address_To_Integer_Pointer is  new  Unchecked_Conversion
        (Source => System.Address,
         Target => Integer_Pointer);

function  Default_Integer_Conversion (Raw_Value: in  System.Address)
    return  Integer is
--|*****************************************************************
--| Description:
--|   Convert from a bit string at a system address to an integer
--|   value.  This is valid for a one CPU configuration
--|   only.
--|
--| Parameter Description:
--|   raw_vlaue   ->  The address of the bit string to convert.
--|
--| Notes:
--|     none
--|*****************************************************************
    Value_Pointer: Integer_Pointer;
    begin
      Value_Pointer := Address_To_Integer_Pointer(Raw_Value);
      RETURN  Value_Pointer.all ;
    end  Default_Integer_Conversion;
pragma  Inline (Default_Integer_Conversion);

--
--  Create the package to convert from bit strings to integers.
--
     package Rtm_Integers is  new  Convert_Integers
      (Width => 15,
       Source_Representation => Integer,
       Target_Conversion => Default_Integer_Conversion);
```

Figure 5-9: Types Conversion Code Fragment

5.5. System Architecture Considerations

There are several points that arise when trying to design an "add-on" system that does not perturb the timing of the original system:

- "You should design the system right in the first place."
- "It can't be done with a software-only approach."
- "You need additional processors to minimize the impact."
- "You need a hardware-only solution which has access to all the address, data, and control lines of the CPU."

Clearly, the ability to design anything perfectly is beyond the scope of human capabilities. The only thing that can make a software-only approach feasible is for the "add-on" system to execute in the background with CPU cycles not needed by the application. Additional CPUs allows us to offload most of the processing from the application CPU, but there is still a small impact on the application processor when its memory is accessed. The final option will be given additional consideration later.

One additional problem which imposes itself on this design is that we have no control over the target hardware for the RTM application system. Therefore, we approached the design with the view that if two (or more) processors are available, the RTM needs a natural breakpoint that can accommodate this. But if everything must execute on one system, it must also operate in this environment. It was partly for this reason and partly to abstract away the application that the *rtm_core* was created. The interface between the *dialogue_manager* and the *rtm_core* is the breakpoint for a multi-processor system.

5.5.1. One CPU

In the one-CPU configuration (shown in Figure 5-10), the RTM and the application are both executing as dependent tasks of a controller application (under control of the Ada run-time system or the host operating system), with the *rtm_core* as part of the monitor. The timing and control of the application knows when there is time available for background processes and suspends itself for a predetermined length of time to allow the RTM to execute.

5.5.2. Two CPUs

In the two-CPU configuration (shown in Figure 5-11), the RTM and the application are executing on different CPUs connected by a DMA hardware link, and the *rtm_core* is a part of the application software. Thus, only the *rtm_core* and the application share address space. The RTM is executing independently and communicating to the user. When the *dialogue_manager* communicates with the *rtm_core*, it is a bus transfer. The concept is the same: a block of commands are formatted and transferred to the *rtm_core*, while the *dialogue_manager* waits for the results. When the application has spare time on its processor, it allows the *rtm_core* to execute. When the *rtm_core* finds commands in its command buffer, it processes them and places the results in the data buffer, sends these results to the RTM, and returns control to the application.

5.5.3. Host-Monitoring Hardware Environment

One interesting variation on the two CPU configuration occurs when the second CPU is not the *rtm_core* running on the application processor, but rather a hardware monitoring device sitting on the address and data lines of the target processor. What this variation can accomplish is the ultimate goal of nonintrusive monitoring with an abstract user interface. Depending on the intelligence of the monitoring hardware (i.e., is it programmable):

- An intelligent hardware monitor can be set up to understand the same commands as the *rtm_core*.
- A dumb hardware monitor can be commanded by modifying the *dialogue_manager* to generate commands in a new format.

Either approach has the advantage of not altering the user interface in any way. All changes are low-level communications changes which are highly insulated from the rest of the RTM.

5.5.4. Host-Multiple Target Environment

Finally, the generalization of the RTM from a two-CPU environment (one for the RTM and one for the application) to a multiple CPU environment (one for the RTM and n for the application) is straightforward. It requires generalizations to:

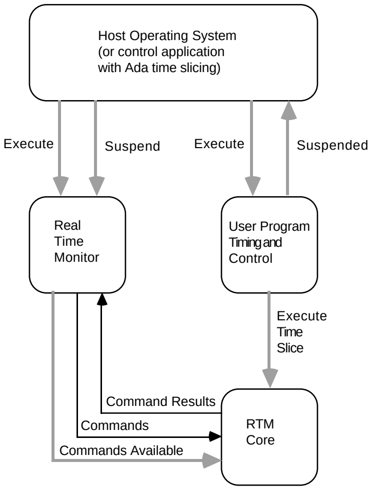

Figure 5-10: One-CPU Configuration

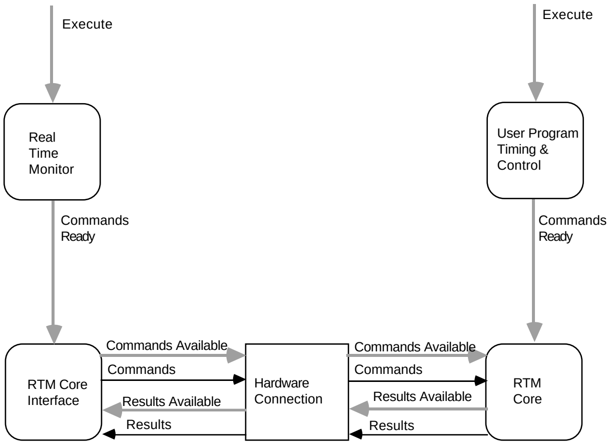

Figure 5-11: Two-CPU Configuration

- The *variable_database* and *library_interface* abstractions to include CPU information.
- The address abstraction to include a CPU designation.
- The *dialogue_manager* so that it knows how to use the CPU information in the address abstraction to communicate with the appropriate processor (for a given command).
- The *types_manager* to convert low-level bit representations from multiple CPUs.

All these changes are in low-level, system-dependent packages and do not impact the basic structure or functionality of the RTM. The ***forms management subsystem*** still runs on a single CPU and interfaces to the user. The *rtm_core* is still part of an application (running on each of the application CPUs) and interfaces to one copy of the RTM, running the user interface.

5.6. Conclusion

The discussions presented above were meant to highlight the troublesome areas encountered while implementing the RTM. Further detail about the implementation and how these items were addressed can be found in the *Prototype Real-Time Monitor: Ada Code* [Van Scoy 87b].

Appendix A: Software Architecture Notation

The notation used in this report to describe software architecture is a modified form of the notation expounded on by Grady Booch in his books on software engineering with Ada [Booch 87a] and reusable software components with Ada [Booch 87b]. The notation used is true to the intent of Booch's notation. The variations (i.e., extensions) are:

- We use reduced package, subprogram, and task icons inside larger icons rather than the object (or blob) icon.
- We use object dependency arrows more subtly, to distinguish different types of dependencies (discussed in Figure A-1 (c)).
- We layered the diagrams, i.e., we show a diagram of top-level dependencies and then expand the bodies of the figures to show the next layers of detail.
- We do not show the internal details of any reusable subsystem, package, subprogram, or task that is used.

One final note about the notation: the figures need not show all the fine-grained detail of a package or subprogram. When the code of a package (or subprogram) is compared to a figure associated with that package (or subprogram), there may be nested procedures or packages not shown on a particular picture, or it may depend on a package not explicitly shown in the figure. The guidelines for these cases are:

- Utility packages or services are not shown (text_io, reusable data structure packages, math libraries, etc.).
- The figures are meant to show the significant details at a particular level, not all the details.
- The definition of "a significant detail" is solely at the discretion of the designer.

With these ideas in hand, Figures A-1 through A-4 explain the meaning of each of the icons available using this notation.

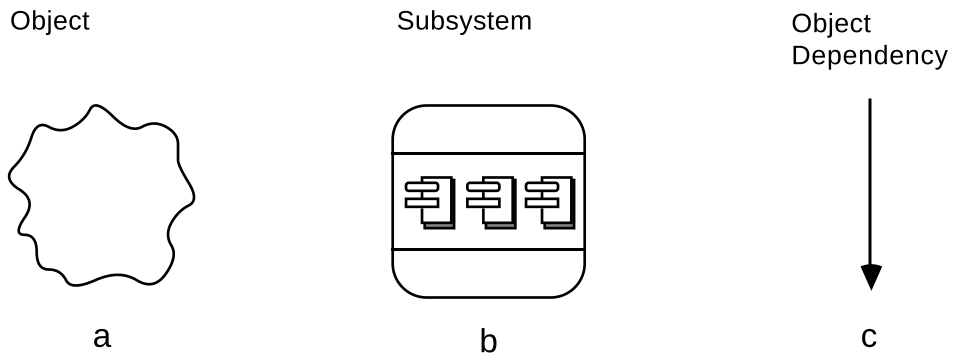

Figure A-1: Object, Subsystem, and Dependency Notation

The object (or blob) icon, shown in Figure A-1 (a), represents an identifiable segment of a system about which we have no implementation information (either by choice or ignorance).

The subsystem icon, shown in Figure A-1 (b), represents a major system component that has a clearly definable interface, but is not representable as a single Ada package.

The object dependency symbol, shown in Figure A-1 (c), indicates that the object at the origin of the arrow is dependent on the object at the head of the arrow. The origin of the arrow indicates where the dependency occurs. If the origin is in the white area of an icon (shown in subsequent figures), it indicates a specification dependency. If the origin is in a shaded area, it indicates a body dependency.

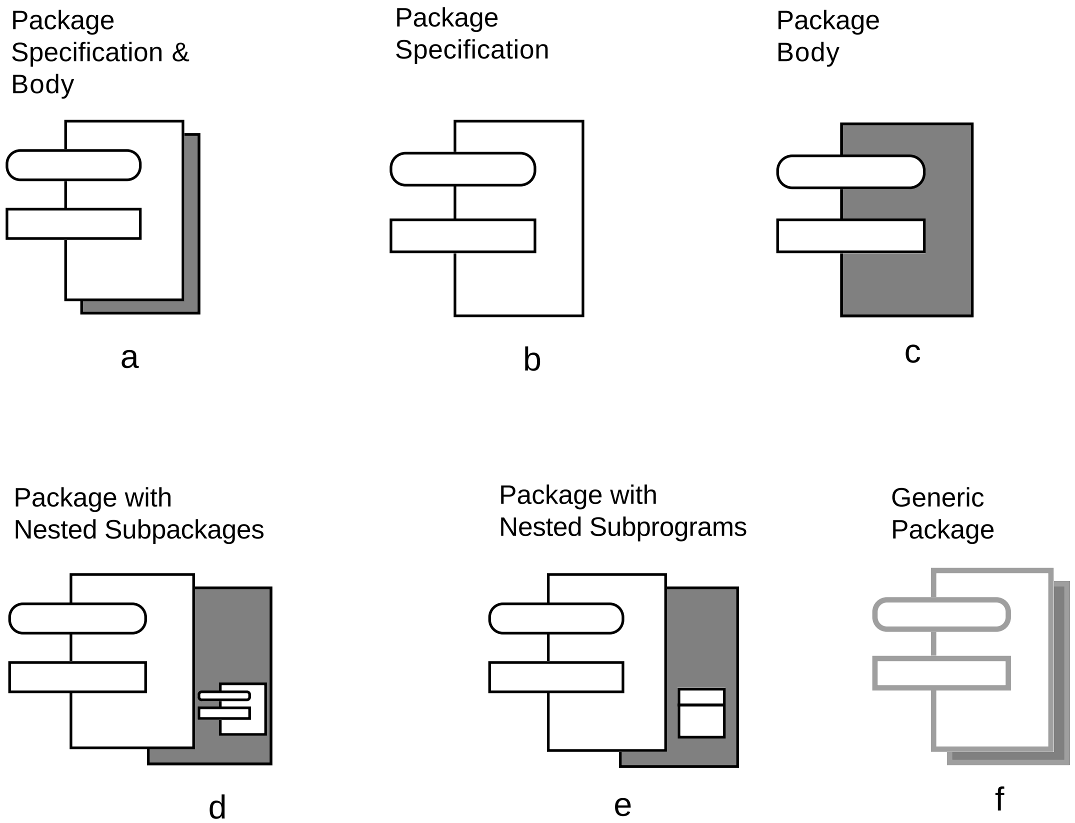

Figure A-2: Package Notation

The package specification and body icon, shown in Figure A-2 (a), represents an Ada package specification (the white area) with an associated package body (the shaded area). This icon can be broken apart to show a package specification, Figure A-2 (b), or a package body, Figure A-2 (c).

Figures A-2 (d) and (e) are variations on the package icon which show greater detail. Figure A-2 (d) is used to represent packages that have nested subpackages within the body; if the small package icon were placed within the specification, it would indicate visible nested packages. Similarly, Figure A-2 (e) illustrates the notation used for separate subprograms within the body of a package.

Finally, Figure A-2 (f) illustrates the icon used for generic packages. Everything discussed above regarding regular packages can also be applied to generic packages.

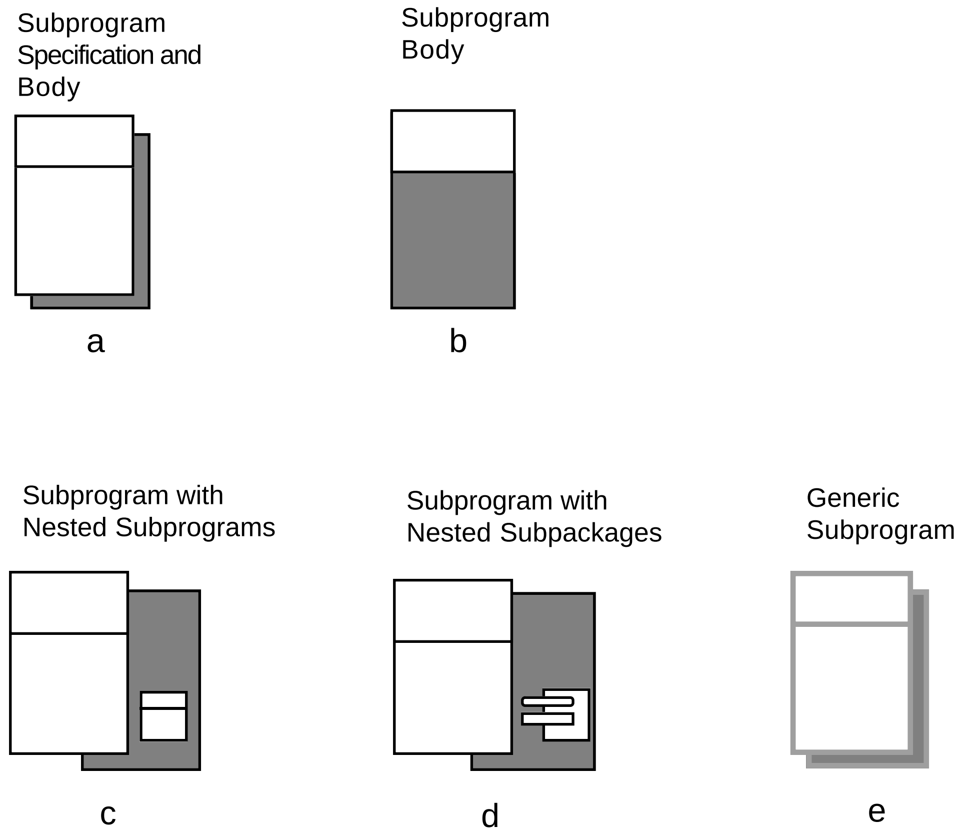
Subprogram Specification and Body
a
Subprogram Body
b
Subprogram with Nested Subprograms
c
Subprogram with Nested Subpackages
d
Generic Subprogram
e

Figure A-3: Subprogram Notation

Much of what was discussed previously regarding packages also applies to subprograms. The subprogram specification and body icon, shown in Figure A-3 (a), represents an Ada subprogram specification (the white area) with an associated subprogram body (the shaded area). This icon can be broken apart to show a subprogram body, Figure A-3 (b).

Figures A-3 (c) and (d) are variations on the subprogram icon which show greater detail. Figure A-3 (c) is used to represent subprograms that have nested subprograms within the body. Similarly, Figure A-3 (d) illustrates the notation used for separate subpackages within the body of a subprogram.

Finally, Figure A-3 (f) illustrates the icon used for generic subprograms. Everything discussed above regarding regular packages can also be applied to generic subprograms.

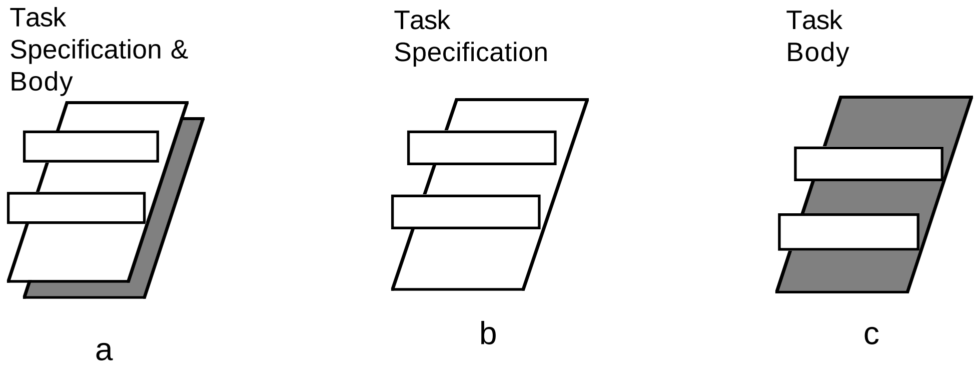

Figure A-4: Task Notation

Again, much of what was discussed previously regarding packages and subprograms applies to tasks. The task specification and body icon, shown in Figure A-4 (a), represents an Ada task specification (the white area) with an associated task body (the shaded area). This icon can be broken apart to show a task specification, Figure A-2 (b), or a task body, Figure A-4 (c). Although they are not shown, nested packages and subprograms are represented in exactly the same manner as shown in Figure A-2 for packages and subprograms.

Appendix B: Data and Control Flow Diagrams

The notation used for data and control flow in this report is a modified form of the notation expounded on by Paul Ward and Stephen Mellor in their book on the design of real-time software [Ward 85]. The notation used is true to the intent of Ward and Mellor's notation. The only variations are:

- use of rectangles with rounded corners for processes
- use of a square for external entities

Aside from these minor cosmetic changes, the data and control flow diagrams used here follow the conventions set forth by Ward and Mellor. We have not developed the pictures to the their fullest extent, but rather used an existing notation to illustrate the thinking involved. Figures B-1 through B-3 briefly explain the symbols available using this notation.

Figure B-1: Store Notation

The data store icon, shown in Figure B-1 (a), represents a place where data are held until needed by a process.

The event store icon, shown in Figure B-1 (a), represents a place where control signals are held until needed by a process.

Figure B-2: Process Notation

The data transformation icon, shown in Figure B-2 (a), represents a process which accepts input data from a data flow(s), control signal(s) from an event flow(s), performs processing on the input data, and transfers the data out over a data flow(s).

The control transformation icon, shown in Figure B-2 (b), represents a process which accepts a control signal(s) from an event flow(s), performs processing on the control signal, and transfers information out over an event flow(s).

The external entity icon, shown in Figure B-2 (c), represents a physical device capable of generating and/or accepting data and control flows.

Figure B-3: Flow Notation

The data flow symbol, shown in Figure B-3 (a), represents the transfer of data from one process to another or to an external entity. This a discrete transfer, i.e., the data are available until read and then no longer available via the flow.

The event flow symbol, shown in Figure B-3 (b), represents the transfer of a control signal from one to another process or to an external entity. This a discrete transfer, i.e., the signal is available until read and then no longer available via the flow.

The time-continuous flow symbol, shown in Figure B-3 (c), represents the transfer of data from one process to another or to an external entity. This a continuous transfer, i.e., there is always data available via this flow. For example, this flow might come from an external monitoring device.

Bibliography

[Ada 83] *American National Standard Reference Manual for the Ada Programming Language,*
ANSI/MIL-STD-1815A-1983, 1983.

[Booch 87a] Booch, Grady.
Software Engineering with Ada.
Benjamin/Cummings, Menlo Park, CA, 1987.

[Booch 87b] Booch, Grady.
Software Components with Ada.
Benjamin/Cummings, Menlo Park, CA, 1987.

[D'Ippolito 87] D'Ippolito, R., K. Lee, C. Plinta, M. Rissman, and R. Van Scoy.
Prototype Real-Time Monitor: Requirements.
Technical Report CMU/SEI-87-TR-36, Software Engineering Institute, November, 1987.

[Texas Instruments 85a]
User Manual for a Form Generator System in Ada.
Equipment Group - ACSL, P.O. Box 801, MS 8007, McKinney, TX 75609, 1985.

[Texas Instruments 85b]
User Manual for an ANSI X3.64 Compatible Virtual Terminal in Ada.
Equipment Group - ACSL, P.O. Box 801, MS 8007, McKinney, TX 75609, 1985.

[Van Scoy 87a] Van Scoy, R., C. Plinta, T. Coddington, R. D'Ippolito, K. Lee, and M. Rissman.
Prototype Real-Time Monitor: User's Manual.
Technical Report SEI-CMU/SEI-87-TR-37, Software Engineering Institute, November, 1987.

[Van Scoy 87b] Van Scoy, R.
Prototype Real-Time Monitor: Ada Code.
Technical Report CMU/SEI-87-TR-39, Software Engineering Institute, November, 1987.

[Ward 85] Ward, Paul T., and Stephen J. Mellor.
Structured Development for Real-Time Systems.
Yourdon Press, Englewood Cliffs, N.J., 1985.

Index

Table of Contents

List of Figures